ISBN: 978-3-95450-191-5 doi:10.18429/JACoW-SRF2017-TUPB069

RIGOROUS DATA PROCESSING AND AUTOMATIC DOCUMENTATION OF SRF COLD TESTS*

K. G. Hernández-Chahín[1†], S. Aull, N. Stapley, P. Fernandez Lopez, N. Schwerg
CERN, Geneva, Switzerland
[1]also at DCI-UG, Guanajuato, Mexico

Abstract

Performance curves for SRF cavities are derived from primary quantities which are processed by software. Commonly, the mathematical implementation of this analysis is hidden in software such as Excel or LabVIEW, making it difficult to verify or to trace, while text-based programming like Python and MATLAB require some programming skills for review and use. As part of an initiative to consolidate and standardise SRF data analysis tools, we present a Python program converting a module containing the collection of all commonly used functions into a LaTeX(PDF) document carrying all features of the implementation and allowing for a review by SRF experts, not programmers. The resulting document is the reference for non-experts, beginners and test stand operators. The module is imported in any subsequent processing and analysis steps like the symbolic analysis of the measurement uncertainties or the study of sensitivities. As an additional layer of protection the functions can be further wrapped including assertions, type and sanity checks. This process maximises function reuse, reduces the risk of human errors and guarantees automatically validated and documented cold test results.

INTRODUCTION

During cold tests of SRF cavities a great number of different signals are measured and automatically recorded. The acquisition and storage is done by test programs often written in LabVIEW [1,2] or INSPECTOR [3]. From these primary quantities we derive figures of merit such as the unloaded quality factor Q_0 and the accelerating gradient E_{acc}. The first processing and derivation of these quantities is done immediately using the data acquisition software, where the routines for the calculations are implemented using the programing language provided by the software manufacture. The final analysis of the data is conducted during the post-processing using Excel [4], MATLAB [5] or Python [6,7] where the analysis routines are custom libraries of the user.

Over the last year we came across the following problems and pitfalls of this approach:

- In LabVIEW and Excel the implementation of the analysis function is somehow hidden from the operator and difficult to verify by the user.
- The functions have to be implemented twice, once for the test software and once for post-processing hence doubling the effort for maintenance.
- Although it is instructive and recommended for any beginner in the field of SRF to implement their own set of analysis functions, for routine operation a single reviewed and protected set is needed.
- In general the implementation of mathematical expressions in any programming language (graphical or text) is cumbersome or unpleasant to read.

In order to overcome these problems the cold-test software system should meet the following requirements:

- Single point of function definition
- Built-in documentation
- Version control
- Nesting and re-use of the core implementation
- Simple and reliable review process
- Automatic tests and sanity checks

In this paper we present an approach consisting of two core components: 1) a central python implementation (`srf_functions`) of all analysis functions as e.g. provided in Padamsee's book [8] and 2) an automatic documentation tool (`dokator`) converting the functions and especially the underlying code into a type-set representation for review by SRF experts and practitioners and not programmers.

SRF_FUNCTIONS

Figure 1 shows the approach with the central element the `srf_functions`. This module or library contains all mathematical expressions used for the evaluation of the measurements and used in the construction of more complex analysis routines. These functions are used for all computations from the data taken with the measurement software (INSPECTOR or LabVIEW) during the cold test to the post processing of the final results.

Single Point of Function Definition

We base our approach on one single Python [6,7] module, `srf_functions`, which contains the implementation of all computations. We chose Python due to its wide spread use, flexibility and various freely available packages. The implementation is kept to a minimum of complexity while placing the effort in clean definitions and documentation.

Un-typed Function Definition

In the core module we define all functions un-typed. This means that the implementation does not rely on any specific features as long as the used objects support standard math operation such as `+ - * / **` with the latter being the power operator in Python. Special symbols or functions like

* Work supported by the Beam Project (CONACYT, Mexico).
† karim.gibran.hernandez.chahin@cern.ch

ISBN: 978-3-95450-191-5 doi:10.18429/JACoW-SRF2017-TUPB069

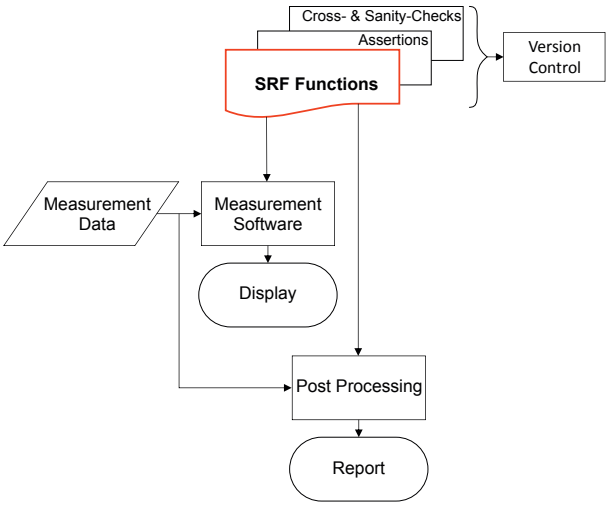

Figure 1: Flowchart of the measurement data from the analysis tools towards the final report with the central definition of all functions in the SRF-function module.

π can be defined in the preamble of the module and may be overwritten on import.

This approach allows to use the functions with any kind of Python objects supporting the aforementioned operators which is true for numbers as well as symbols as provided by the sympy package [9, 10].

For the construction of more complicated or coupled expressions the functions can be called and re-wrapped into higher order systems. This, e.g., allows to automatically derive the expressions for the loaded Q and accelerating gradient based only on primary quantities:

$$\left[\begin{array}{c} Q_0 \\ E_{acc} \end{array}\right]_{(P_f,P_r,P_t,f,k,\kappa_E)} = \left[\begin{array}{c} \frac{2\pi P_t f k}{P_f - P_r - P_t} \\ \sqrt{\frac{P_t k}{\kappa_E}} \end{array}\right] \quad (1)$$

Built-In Documentation

A very useful Python feature are docstrings [11]. A docstring is the first statement in a function or class declaration providing the user with the documentation. This can be used to include all information necessary to understand the function call signature, the physics behind the computation as well as give reference to text books such as [8].

Maintaining the following conventions the docstring can be made human and computer readable allowing for automatic processing and display. The documentation consists of the following blocks:

- Description of the greater context of the function (optional).
- Definition of the parameters with their symbol, name and unit.
- Description of the return value(s) with name, symbol and unit.
- Optional notes for function use.
- References used in the description and/or notes.
- Examples illustrating the function use, common results and providing a layer of tests.

Furthermore variable names may use LaTeX notation such as underscores for subscripts.

Notice that the approach is similar to common documentation tools such as Epydoc [12] and also bases on features of `reStructuredText` [13]. An example of the built-in documentation is shown in Fig. 2 highlighted in red.

Version Control

The `srf_function` module and its documentation are included in a version control system such as EDMS [14] or GitLab [15] in order to keep track of all changes and making the current version available to the team. Back-tracing problems and evaluating improvements is aided by the open source of the Python module and text-based comparison of different versions of the code.

Automated Testing

Software engineering encourages the use of various testing techniques as part of a software quality assurance testing strategy to maintain the required level of reliability, reduce the risk of failure during operation, and provide confidence that code changes will not have any adverse affects [16].

When testing, different types of defect are found at each level of abstraction, and the lowest level, code (unit) tests, are often implemented to ensure correctness for individual functions as a minimum testing requirement.

For this Python provides docstring which serves as both simple examples of the use of the function in the documentation, and for function verification upon module import using doctest [17]. Doctest's power rests in its simplicity [18, p 157], but for larger more robust test scenarios unittest is preferred.

Sanity Checks

Extending on the possibility to import and re-use the core definitions the functions can be wrapped by a facade layer [19] to include typing and sanity checks. This ranges from enforcing certain data types, including assertions [20] for certain inputs such as power reading shall be greater than a certain minimum or zero, as well as proper error handling to prevent the process from crashing.

LabView Import

LabVIEW is often used for cavity testing. Hence it is advantageous to use the same math functions for the post-processing and for the actual cavity tests. There are several general solutions for including Python functions in LabVIEW, each with its own advantages in terms of ease, and performance. We describe three, just to demonstrate the possibilities and provide an idea of the benefits of each.

One of the simplest solutions is to use LabVIEW 's built-in ability (`System Exec VI`) to execute a Python script containing the math functions [21]). A new instance of the Python script is executed every time, meaning this solution is comparatively slow, taking more than 100 milliseconds on

ISBN: 978-3-95450-191-5 doi:10.18429/JACoW-SRF2017-TUPB069

a “standard PC”. Furthermore, its necessary to develop and maintain processing code to parse the string output from the function into something meaningful for LabVIEW. However, performance concerns can be mitigated, for example, by passing sets of data as arguments to a function and which could return an array of results to replace repeated function calls.

For faster performance, especially for scenarios where many repetitive calls to Python functions are required, another solution would be to reduce the overhead of starting a Python program for every call. This is typically implemented by putting the functions in a Python server program and then communicating with it by using some type of inter-process communication such as simply a TCP socket, or some facilitation mechanism like a message broker (see for example using LabVIEW with ZeroMQ [22] or [23]). The inconvenience is that the server program must be managed and started before LabVIEW can communicate with it, and again the inputs and outputs must be parsed in a similar way to the first solution.

Probably the most performant and efficient solution is by providing in-process function access within LabVIEW by using a library to encapsulate the starting of a Python interpreter and function execution. The conversion of basic types between Python and LabVIEW could be included too. However, this is not so easy to do so, meaning development time is longer, and the library is platform specific. Be aware that an incorrectly written library can cause instabilities or failure for the host process (LabVIEW). A careful judgement has to be made if the investment is worth it.

Finally convenient COTS (Commercial off-the-shelf) solutions do exist such as that from Enthought [24] which facilitates easy use of Python functions within LabVIEW , reducing much of the effort required in the previous solutions. Enthought provides automated type conversion between LabVIEW and Python, good documentation, and working examples [25].

DOKATOR

The review of the implementation of computer programs requires at least some programming skills. If however, the mathematical expressions are displayed in their type-set representation, the implementation can be reviewed and approved by any SRF practitioner and expert.

We wrote a small program `dokator` allowing to read the content of a well formatted module and automatically converting it into a PDF using LaTeX [26] or a webpage using MathJax [27]. The program displays the entire built-in documentation including the examples (tests) and type-sets the actual implementation.

An example of the Python input and the compiled LaTeX output is shown in Fig. 2.

From the function signature we can automatically create mathematical symbols for all arguments using sympy [9, 10] and execute the un-typed function resulting in a symbolic expression. The symbolic expression is displayed as LaTeX.

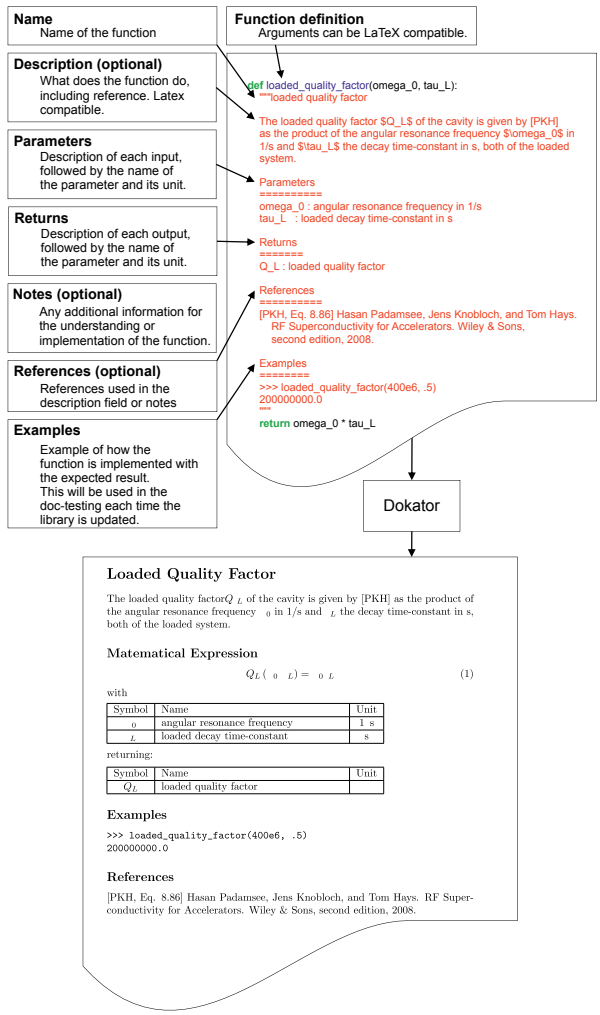

Figure 2: Example for a input function in Python and LaTeX output as produced by the dokator.

Any mistake in the implementation is then converted to LaTeX and is consequently visible to the reviewer, including the possibility to catch inconsistencies in the definition on the arguments and the description of them in the documentation.

Furthermore, sympy can infer some type-setting options from the used variable names. So it would replace known LaTeX keywords such as `eta` by η and anything following the first underscore _ by a subscript.

The software review can now be done either on the source code or on the converted documentation. The program `dokator` provides a verifiable mapping between the two representation. The symbolic representation of the mathematical functions provides further advantages for the analysis of the SRF cold test system. Since the derivatives of the functions can be computed analytically this allows, e.g., for the analytical study of sensitivities or the analysis of error propagation after GUM [28].

CONCLUSIONS AND OUTLOOK

Due the nature of SRF cold tests where the cost involved, time for preparation and availability of helium are significant

ISBN: 978-3-95450-191-5 doi:10.18429/JACoW-SRF2017-TUPB069

factors, a reliable and robust framework for the visualisation and analysis of the measurement data is needed. Furthermore, the approach needs to be flexible to allow for modification and experiments during the measurement campaign while remaining verifiable, traceable and transparent.

In this paper we have outlined a software approach addressing the problems of transparency, reliability, review and traceability of the test software during cold test and analysis. The approach is based on Python and a number of great and free software tools / Python packages. With every of the proposed measures the measurement process becomes more robust and the operator gains time to concentrate on the object under test - instead of debugging the system.

ACKNOWLEDGEMENT

Special thanks to Alick Macpherson, Alejandro Castilla and Katarzyna Turaj for feedback.

REFERENCES

[1] National instruments: Labview. [Online]. Available: http://www.ni.com/en-us/shop/labview.html

[2] G. W. Johnson and R. Jennings, *LabVIEW Graphical Programming*, 4th ed. McGraw-Hill Book Company, Inc., 2006.

[3] B. Lefort and M. Ferrari, "Inspector user manual - a rapid development application for cern instrumentation," CERN Manual, April 2014. [Online]. Available: https://espace.cern.ch/AD-site/Inspector/inspector%20User%20Manual.pdf

[4] Microsoft: Excel. [Online]. Available: https://products.office.com/en-us/excel

[5] Mathworks: Matlab. [Online]. Available: https://www.mathworks.com/products/matlab.html

[6] Python software foundation. [Online]. Available: https://www.python.org/

[7] (2013, December) pythonxy - scientific-oriented python distribution based on qt and spyder. [Online]. Available: http://code.google.com/p/pythonxy/wiki/Welcome

[8] H. Padamsee, J. Knobloch, and T. Hays, *RF Superconductivity for Accelerators*, 2nd ed. Wiley & Sons, 2008.

[9] (2013, December) Sympy. [Online]. Available: http://sympy.org/en/index.html

[10] "Sympy documentation," SymPy Development Team, November 2014.

[11] Pep 257 – docstring conventions. [Online]. Available: https://www.python.org/dev/peps/pep-0257/

[12] Epydoc - automatic api documentation generation for python. [Online]. Available: http://epydoc.sourceforge.net/

[13] R. Jones, "A ReStructuredText Primer," docutils.sourceforge.net, March 2013. [Online]. Available: http://docutils.sourceforge.net/docs/user/rst/quickstart.html

[14] Cern: Engineering data management service (edms). [Online]. Available: http://edms-service.web.cern.ch/edms-service/faq/EDMS/pages/

[15] Gitlab. [Online]. Available: https://about.gitlab.com/

[16] S. et al., *Software Testing Foundations*, 4th ed. RockyNook Inc., 2014.

[17] "doctest - test interactive python examples," Python Documentation, 2.7.13. [Online]. Available: https://docs.python.org/2/library/doctest.html

[18] A. Maxwell, *Powerful Python*, 2nd ed. Powerful Python Press, 2017.

[19] Facade pattern. [Online]. Available: https://en.wikipedia.org/wiki/Facade_pattern

[20] Using assertions effectively. [Online]. Available: https://wiki.python.org/moin/UsingAssertionsEffectively

[21] *Call Perl and Python Scripts from LabVIEW*, National Instruments, March 2016. [Online]. Available: http://www.ni.com/tutorial/8493/en/

[22] zeromq - super socket bindings for labview. [Online]. Available: http://labview-zmq.sourceforge.net/

[23] Python-labview communication. [Online]. Available: https://github.com/Kricki/py-lv-comm

[24] Enthought, inc. [Online]. Available: https://www.enthought.com/

[25] *Python Integration Toolkit - Examples*, Enthought, 2016. [Online]. Available: http://docs.enthought.com/python-for-LabVIEW/guide/examples.html

[26] L. Lamport, *LATEX: a document preparation system : user's guide and reference manual*. Addison-Wesley Pub. Co., 1994, no. p. 2.

[27] American mathematical society mathjax consortium. [Online]. Available: https://www.mathjax.org/

[28] "Evaluation of measurement data — guide to the expression of uncertainty in measurement," JCGM 100:2008, 2008. [Online]. Available: http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf